Express Mail Label No. EL994264374US

IN THE UNITED STATES PATENT AND TRADEMARK OFFICE
APPLICATION FOR UNITED STATES LETTERS PATENT

APPLICATION MONITOR SYSTEM AND METHOD

By:

Joseph G. Laura
7620 Tensley Drive
Plano, Texas USA 75025
Citizenship: USA

Application Monitor System and Method

CROSS-REFERENCE TO RELATED APPLICATIONS

[0001] None.

STATEMENT REGARDING FEDERALLY SPONSORED RESEARCH OR DEVELOPMENT

[0002] Not applicable.

REFERENCE TO A MICROFICHE APPENDIX

[0003] Not applicable.

FIELD OF THE INVENTION

[0004] The present invention is directed to computer software, and more particularly, but not by way of limitation, to a system and method for monitoring a computer application.

BACKGROUND OF THE INVENTION

[0005] In the realm of system and software operation and development, it is sometimes useful to monitor applications during operation. Software applications, computer programs, software subroutines, software modules, or software components (hereinafter referred to as applications) may write intermediate information to internal variables which are not output by the application. Because the content of these intermediate variables is not output, the content of these variables is not generally accessible to testers or other programs. For this reason, it may be difficult, for example, to debug applications operating erroneously without sufficient understanding of these intermediate variables or stages of

processing of the application. Monitoring applications and these intermediate variables may be warranted for a number of other reasons.

SUMMARY OF THE INVENTION

[0006] The present disclosure provides a system for monitoring an application. The system comprises a first module operable to read application values stored in a memory area by the application, a second module in communication with the first module and operable to request the first module to read the application values, the second module further operable to receive the application values from the first module, a third module in communication with the second module, the third module operable to display the application values.

[0007] In another embodiment a method of monitoring operation of an application is provided. The method comprises running an application, generating application values stored in a memory area, reading the memory area used by the application to obtain the application values, and displaying the requested application values.

[0008] In another embodiment a system for non-intrusively monitoring variables during operation of an application is provided. The system comprises a compile listing having an address map with an offset for at least one variable of an application and a module operable to read the compile listing and obtain the offset of the at least one variable of the application, the module further operable to attach to a memory area where the application is operating to obtain a value for the variable using the offset.

[0009] These and other features and advantages will be more clearly understood from the following detailed description taken in conjunction with the accompanying drawings and claims.

BRIEF DESCRIPTION OF THE DRAWINGS

[0010] For a more complete understanding of the present disclosure and the advantages thereof, reference is now made to the following brief description, taken in connection with the accompanying drawings and detailed description, wherein like reference numerals represent like parts.

[0011] Figure 1 is a block diagram of an application monitor system according to one embodiment.

[0012] Figure 2 illustrates a shared memory routine for enabling shared memory in COBOL programs.

[0013] Figure 3 illustrates a socket routine for enabling socket communications in COBOL programs.

[0014] Figure 4 is a block diagram of an application monitor system according to another embodiment.

[0015] Figure 5 illustrates an exemplary general purpose computer system suitable for implementing the several embodiments of the application monitor system.

DETAILED DESCRIPTION OF THE PREFERRED EMBODIMENTS

[0016] It should be understood at the outset that although an exemplary implementation of one embodiment of the present disclosure is illustrated below, the present system may be implemented using any number of techniques, whether currently known or in existence. The present disclosure should in no way be limited to the exemplary implementations, drawings, and techniques illustrated below, including the exemplary design and implementation illustrated and described herein.

[0017] It is often desirable to access the values of internal variables of applications. The software developer may desire to view these internal variable values to help understand and fix the operation of his application during the testing phase. A business analyst may want to have access to internal variables to optimize the interactions among multiple independent applications. Generally, however, these values are not accessible.

Shared Memory Embodiment

[0018] Turning now to Figure 1 a block diagram of an application monitor system 10 for accessing the value of internal application variables is depicted. A first application 12 is programmed or coded to create a first shared memory 14 and to write values into variables var1 16a, var2 16b, and varN 16c contained within the first shared memory 14. In addition, the first application 12 is coded to perform the task allocated to the first application 12. In the present embodiment, the application 12 runs in a normal, real-time operational mode, as opposed to running under a debugger or other tool employed to encapsulate the application 12. While the application 12 is in real-time operation, whether during testing or other simulated conditions, or otherwise running, a monitor module 18 is operable to read

the values of the variables 16 from the first shared memory 14 and is in communication with a client 20. A second application 22 is programmed or coded to create a second shared memory 24 and to write values into variables varA 26a, varB 26b, and varX 26c contained within the second shared memory 24. The monitor module 18 is operable to read the values of the variables 24 from the second shared memory 24.

[0019] A user interface 28 is in communication with the client 20. The user interface 28 is operable to select the applications 12, 22 and to select the variables 16, 24 and to communicate these selections to the client 20. The client 20 requests the values of the variables 16 and 26 stored in the first and second shared memory 14 and 24, based on the variables 16 and 26 selected by the user interface 28. The client 20 receives the values of these variables 16 and 26 from the monitor module 18 and communicates the values of these variables 16 and 26 to the user interface 28. The user interface 28 is further operable to display the values of variables 16, 26 for presentation to a user. Note that while two applications are depicted in Figure 1, any number of applications executing on the computer system 30 may be concurrently monitored by the monitor module 18. Also note that while the client 20 is depicted communicating with only one monitor module 18, the client 20 may communicate concurrently with multiple monitor modules 18.

[0020] In some embodiments the client 20 may be programmed to periodically request the value of a variable and to send this value to the user interface 28. In this case, the user interface 28 may designate to the client 20 a variable for monitoring, and the client 20 may repeatedly request the value of the designated variable from the monitor module 18 at some periodic rate, for example once per second or ten times per second.

[0021] The applications 12, 22 and the monitor module 18 are computer programs or applications which execute on a general purpose computer system 30. The shared memories 14, 24 are located in the random access memory or in the secondary storage of the general purpose computer system 30. The client 20 and the user interface 28 are computer programs or applications which execute on a general purpose computer system 32. General purpose computer systems will be discussed in more detail hereinafter.

[0022] The application 12 is programmed so when it runs it first creates the first shared memory 14 and then writes some variables of interest out to variables 16 in the first shared memory 14 whenever the application changes the value of those variables 16. The action of the application 12 writing these values to the variables 16 in the first shared memory 14 does not interfere with the normal operation of the application 12. These variables 16 may include an identifier, such as a variable name, and may have a value associated with the identifier. Both the identifier and value of each variable 16 are made available for display on the UI 28 through the application monitor system 10 through the monitor module 18 reading these values and identifier of each variable 16 from the first shared memory 14, the monitor module 18 sending these values and variable names to the client 20, and the client 20 sending these values and variable names to the user interface 28. If the application 12 does not write an internal variable to the first shared memory 14, then this internal variable is not accessible to the monitor module 18 and hence not accessible to the client 20. For example, if EmployeeSocialSecurityNumber is an internal variable of the application 12 and the application 12 does not write the value of EmployeeSocialSecurityNumber to the

first shared memory 14, then the monitor module 18 may not be able to access the value of EmployeeSocialSecurityNumber.

[0023] The user interface 28 may make various uses of the data access provided by the application monitor system 10. This data may be periodically written to a database or to a file. The data may be employed to generate one or more graphical depictions of current operating conditions including bar charts and pie charts. The data may be employed to track and display a "heartbeat" indication of the health of an application. Some data may be monitored and used as a trigger by other applications to take action. The user interface 28 may be used interactively to select variable values for display and to display those selected variable values.

[0024] In some embodiments the client 20 may send the variables 16, including the variable name and variable value, to the user interface 28 in extensible markup language (XML) format. XML supports flexible formatting of the variable names and variable values for transmission to the user interface 28.

[0025] The client 20 and monitor module 18 may communicate using a socket connection. A socket is a communication mechanism which is provided by some operating systems to support communications between processes located on the same computer system and between processes located on separate computer systems. Typically, one party to a socket connection is allocated the responsibility of being ready at any time to receive a socket connection request and to establish the socket connection in response to this connection request. This party to a socket connection is called the socket server. The second party to the socket connection requests a socket connection from the socket server

at any time and is called the socket client. The socket server may serve multiple concurrent socket connection requests. The socket server may establish and maintain several concurrent socket connections with different socket clients. Operating systems which support sockets typically provide a set of system calls which provide mechanisms to control socket connections. When sockets are employed, the monitor module 18 acts as a socket server and the client 20 acts as a socket client.

[0026] In some embodiments the monitor module 18 and the client 20 may be programmed in the Common Business Oriented Language (COBOL) programming language. Unfortunately COBOL may not provide support for sockets or for shared memory. In this case some embodiments may employ a technical layer which provides a means for COBOL programs to avail themselves of certain features, including sockets and shared memory, which otherwise would not be accessible to these COBOL programs. The technical layer in this example provides a plurality of routines including a shared memory routine and a sockets routine. The shared memory routine provides the COBOL program with the functionality to enable memory sharing on the computer 30 between applications, for example between the applications 20, 22 and the monitor module 18. The socket routine enables the COBOL program to communicate information between applications on the same computer and between applications on different computers, for example between the monitor module 18 and the client 20.

[0027] In addition to providing convenient means for COBOL programs to access operating system features, the technical layer isolates the COBOL programs which invoke technical layer routines from changes in the underlying operating system. The technical

layer may be separated into multiple layers including a layer having various operating system calls for compatibility with specific operating systems, such that only a portion of the technical layer would be replaceable to enable compatibility with other operating systems. This would simplify the use of alternate versions of the technical layer for multiple operating systems by abstracting the specific calls to a single layer while leaving the other interface layers facing the COBOL programs the same. This provides for changes in the technical layer to accommodate different operating systems which would be transparent to the COBOL programs using the technical layer. For additional information related to employing the COBOL technical layer and the role of the shared memory routine and the socket routine see U.S. Patent Application Serial No. 10/696,895, filed October 30, 2003, entitled "Implementation of Distributed and Asynchronous Processing In COBOL" by the above-named inventor, incorporated herein by reference for all purposes.

Technical Layer Shared Memory Routine

[0028] Figure 2 illustrates one embodiment of the shared memory routine 50 for sharing memory between COBOL programs. The shared memory routine 50 is provided with an index 52 for maintaining a plurality of keys 54 related to an address 56 of the memory 58 which is utilized as shared memory by the first and second COBOL programs 60 and 62. The first COBOL program 60 requests shared memory from the shared memory routine 50. The shared memory routine 50 includes a plurality of functions operable for managing shared memory, including a create function for creating a shared memory block. According to one embodiment, the shared memory routine 50 is a COBOL program which

issues an operating call to the operating system 64 to allocate memory for sharing between COBOL programs.

[0029] Where the shared memory routine 50 is provided as a COBOL library, the first COBOL program 60 and the shared memory routine 50 may be compiled to enable this functionality. The amount of memory needed by the first COBOL program 60 is designated by the first COBOL program 60, at compile time, and the operating system 64 allocates the memory required for the working storage section of the first COBOL program 60. The operating system obtains the address 56 for the designated memory 58 to be shared between the COBOL programs. Once the shared memory has been identified, the first COBOL program 60, using an identifier, such as an alphanumeric identifier or name, in some embodiments, via a request to the shared memory routine 50.

[0030] The shared memory routine 50 maintains the identifier and the index 52 associated with the key 54. The shared memory routine 50 uses the key 54 to issue a request to the operating system for the associated address 56 of the memory 58. The operating system 64 passes the address 56 back to the shared memory routine 50, which in turn passes the address 56 back to the linkage section 66a of the first COBOL program 60.

[0031] The linkage section 66a of the first COBOL program 60 is typically used for passing information between subprograms or calling programs, but is employed here to map to the address 56 of the memory 58 that is used for shared memory. Mapping the address 56 to the linkage section 66a of the first COBOL program 60 is useful since shared memory only needs to be loaded one time and does not require constant refreshing. For

example, data space is a section of mainframe memory that stays resident when programs exit, but requires a task running to keep the memory refreshed. Core loads are another example of memory employed in mainframe computer systems, but when the program using the memory exits or closes, the memory is released and the data is no longer available. Employing the shared memory routine 50, the memory is maintained by the shared memory routine 50 even after the first COBOL program 60 terminates.

[0032] The second COBOL program 62, using the identifier such as the name, requests the address 56 of the memory 58 wherein the shared memory is located. The shared memory routine 50 returns the address 56 in a similar manner based on the use of the same identifier or name. The address 56 of the shared memory is mapped back to the linkage section 66b of the second COBOL program 62 thereby creating a shared block of memory useable by both the first and second COBOL programs 50 and 52.

[0033] The shared memory routine 50 is efficient since the address 56 of the array to the shared memory block is used only once and may be accessed by a plurality of COBOL programs via the shared memory routine 50. In addition to those routines previously described, the shared memory routine 50 includes additional functions, such as an attach function to attach to an existing block of memory, a detach function to detach from an existing block of memory, a remove function to remove a shared memory block from the system, and a query function to determine whether a shared block of memory exists, the size of the shared memory block, and additional information with regard to the shared memory block maintained by shared memory routine 50.

[0034] In some aspects, the shared memory routine 50 may be provided with additional indexes 100 maintaining identifiers other than alphanumeric or name identifiers or other types of identifiers or only a key queryable by the COBOL programs. In some aspects, the shared memory routine 50 monitors the shared memory when the memory is protected to prevent the shared memory from being overwritten. In other aspects, the memory is unprotected and the shared memory routine 50 allows the COBOL programs to read and write from memory in unrestricted manner.

[0035] In some aspects the shared memory routine 50 manages the protected and unprotected aspects of the shared memory by virtue of the specific operating system call, whereby the operating system 64 manages the protection of the block of memory 58 used for shared memory.

Technical Layer Socket Routine

[0036] Figure 3 is a diagram illustrating the functionality of the socket routine 90 for enabling the COBOL program 92 for communication with a socket 94. The socket 94 may represent a communication channel established between one or more computer systems, such as the computer 30 and the computer 32 (see Figure 1). Functionally, the COBOL program 92 requests communication, via a socket, from the socket routine 90. The socket routine 90 establishes the socket connection with the socket 94, such as by a call to an operating system 64. The operating system 64 may be any operating system operable on the computer 30 or 32. In the present embodiment, the COBOL program 92 is operable on an IBM mainframe computer system using a zOS operating system.

[0037] For example, the COBOL program 92 may initiate an operating system 64 call to an operating system function to accept a connection request from a client. An example of such an operating call is provided in the operating system's callable services reference manual. It may be necessary to refer to specific documentation for the operating system 64 where the technical layer and COBOL program 92 will be used to determine the correct syntax and functionality. The operating system calls, including the syntax and techniques for the COBOL program 92 to communicate with the operating system 64 via the technical layer, may vary greatly depending upon the desired operating system 64. For a more complete understanding of socket communications in COBOL see U.S. Application Serial No. 10/696,895, filed October 30, 2003, entitled "Implementation of Distributed and Asynchronous Processing In COBOL" which, as mentioned above, is incorporated herein by reference.

[0038] Once the socket routine 90 establishes the connection with the socket 94, via the operating system 64, the socket routine is then operable to read and write information from the socket 94. The socket routine 90 receives information from the socket 94 and writes the information to a memory 58, such as a block of memory of the computer 30, 32. In one aspect, such as on a mainframe, the socket routine 90 receives the information from the socket and is operable to convert the information, depending upon the format useful for the particular platform. For example, the socket routine 90 may receive the information in ASCII format and convert the information to EBCDIC format for use on a mainframe, or vice-versa.

[0039] The COBOL program 92 then reads the information from the memory 58 thus enabling the COBOL program 92 to communicate via the socket 94. In one aspect, the COBOL program 92 allocates the memory 58, and thus obtains the address of the memory 58, while in other aspects the socket routine 90 establishes the memory 58 and the COBOL program 92 obtains the address of the memory 58 from the socket routine 90. In either case, the COBOL program 92 employs the address of the memory 58 and lays a map over the memory 58 to read the information from the memory 58. According to one aspect, the COBOL program 92 may employ a copybook for reading the memory 58 information into the COBOL program 92.

[0040] In one embodiment, the socket routine 90 creates the socket 94 and may be thought of as providing a file descriptor that describes a stream. The socket routine 90 obtains or is provided the address or the target where the data or information coming off the socket 94 should be provided. The COBOL program 92 maps the address of the memory location 58 to the working storage section of the COBOL program 92 and, thereafter, the COBOL program 92 can analyze the information obtained from the socket 94 in any desirable manner. The previous example illustrates the additional functionality and flexibility provided by this socket routine 90 to enable COBOL programmers the ability to do messaging in a distributed environment, for example.

[0041] To accomplish writing to the socket 94, the COBOL program 92 may, in one embodiment, write the information or data to the memory 58 or provide the information directly to the socket routine 90. The socket routine 90 obtains the information and writes, via the operating system 64, the information to the socket 94. To accomplish the various

socket 94 related functions, the socket routine 90 is provided with a number of functions callable from the COBOL program 92 or enabled by the technical layer, including a create function to create a new socket, an attach function to attach to existing sockets, and an open function to open the socket 94.

[0042] Other functions in the socket routine 90 include a write function to write data into the socket 94 and a block function to prevent writing when the socket 94 is full. A read function reads data from the socket 94, in a manner similar to that described above. A remove function enables the socket routine 90 to remove sockets 30 from the system, and a delete function is operable to delete sockets where the socket 94 has not yet been opened. As previously discussed, a number of the socket functions described above may be enabled via operating system 64 or other calls from the socket routine 90 to enable the COBOL program 92 for communications via the socket 94. In one aspect, the socket routine 90 may include a listening port function that allows the socket routine 90 to monitor the socket 94 for communications and additional functionality for managing the connection of the socket 94.

[0043] Using the application monitor system 10 described above, the internal states of applications 12, 22 can be easily monitored via the user interface 28. The user interface 28 requests the value of a variable, for example the value of variable var1 16a associated with the first application 12, from the client 20. The client 20 communicates over the socket connection with the monitor module 18 to request the value of a variable, for example the value of var1 16a. The monitor module 18 reads the first shared memory 14 to obtain the value of a variable, for example the value of var1 16a and returns this value to the client

20. The client 20 returns this value to the user interface 28 which displays this value. The application monitor system 10 may be implemented in any programming language and is not limited to being implemented in COBOL. Similarly, the applications 12, 22 may be implemented in any programming language and are not limited to being implemented in COBOL.

Memory Attachment Embodiment

[0044] Turning now to Figure 4 another embodiment of the application monitor system 10 for accessing the value of internal application variables is depicted. A monitor module 18 is in communication with a client 20 which requests the values of variables 110, such as var1 110a and var2 110b, from the first application 12. The monitor module 18 is operable to attach to a first memory area 112 of the first application 12 and to read the values of the variables 110 from the first memory area 112. The monitor module 18 sends the values of the variables 110 to the client 20. The first application 12 writes values into the first memory area 112 during the normal course of executing.

[0045] The client 20 may also request the values of variables 116, such as var3 116a, from the second application 22. The monitor module 18 is also operable to attach to a second memory area 114 of the second application 22. The second application 22 writes values into the second memory area 114 during the normal course of executing. The monitor module 18 is operable to read the values of the variables 116 from the second memory area 114 and to send these values to the client 20. Note that while two applications are depicted in Figure 4, any number of applications executing on the

computer system 30 may be concurrently monitored by the monitor module 18. Also note that while the client 20 is depicted in communication with only one monitor module 18 in Figure 4, the client 20 may communicate concurrently with multiple monitors 18.

[0046] A user interface 28 is in communication with the client 20. The user interface 28 is operable to select the applications 20, 22 and to select the variables 110 and 114 and to communicate these selections to the client 20. The user interface 28 is further operable to receive the values of variables 110 and 114 from the client 20 and to display these values.

[0047] The applications 20, 22 and the monitor module 18 are computer programs or applications which execute on a general purpose computer system 30. The client 20 and the user interface 28 are computer programs or applications which execute on a general purpose computer system 32. General purpose computer systems will be discussed in more detail hereinafter.

[0048] Applications, such as the first application 12 and the second application 22, are typically compiled and linked to create a loadable object or image file which is executable. When the executable is run on the computer system, such as general purpose computer system 30, the application is loaded into the computer system memory. Typically the application is loaded into memory in three sections including an instruction section, a data section, and a stack or dynamic memory section. These sections may or may not be contiguous in memory. The instruction section comprises the computer instructions encoded by the application. The data section comprises memory reserved for variables defined in the application. The stack section comprises a block of memory which the application may dynamically allocate, use, and de-allocate during execution.

[0049] The applications 20 write into the memory areas 112, 114 while executing. Note that in this embodiment it is not necessary for the applications 20, 22 to mirror the values of internal variables by writing duplicate values in shared memory regions 18, 26. In this embodiment the applications 20, 22 do not need to be changed to enable the operation of the application monitor system 10. This aspect of this embodiment is useful if different user communities employ the monitor at different stages of the software lifecycle. For example, the variables which the original software developer may be interested in during the test phase may not be those that a business analyst is interested in when the software is deployed and in service. Because all the application 12, 22 variables are accessible in the memory areas 112 and 114, the needs of various user communities are readily satisfied.

[0050] When the client 20 requests variables 110 from application 12, the monitor module 18 attaches to the memory area 112 of the application 12. Typically the monitor module may attach to the memory area 112 by executing operating system calls. The monitor module 18 may remain attached to the memory area 112 after completing the request of the client 20, or the monitor module 18 may un-attach after completing the request of the client 20. In some embodiments a memory manager may be employed to provide the monitor module 18 with information needed to attach to the memory areas 112 and 114, for example the first address and the total size of the memory areas 112 and 114.

[0051] A first address map 116 is stored in secondary storage 118. This secondary storage may be closely associated with computer system 30, with computer system 32, or with some other remote computer system. The first address map 116 contains a listing of

all variables names associated with the application 12 and their address offset relative to the first instruction in the application 12 executable file. The address offset is used to generate an address to read the value of the variable from the memory area 112. Similarly, a second address map 120 is stored in secondary storage 118 which contains a listing of all variable names associated with the application 22 and their address offset relative to the beginning of the application's 22 address space. The address maps 116 and 120 may be generated when compiling and linking the applications.

[0052] In some embodiments the client 20 accesses the secondary storage 118, reads the address map 116 or 120, identifies the address offset associated with a selected variable, and sends the address offset to the monitor module 18 when requesting a variable value. In this case, the monitor module 18 reads the value at the address offset in the memory area 112 or 114 and returns the value to the client 20. In other embodiments the monitor module 18 receives a client 20 request for the value of a variable, wherein the client 20 names the variable, the monitor module 18 accesses the secondary storage 118, reads the address map 116 or 120, identifies the address offset associated with the variable name, reads the value at the address offset stored in the memory area 112 or 114, and returns the value to the client 20.

[0053] As discussed above under the shared memory embodiment, the user interface 28 may make various uses of the data access provided by the application monitor system 10. This data may be periodically written to a database or to a file. The data may be employed to generate one or more graphical depictions of current operating conditions including bar charts and pie charts. The data may be employed to track and display a

"heartbeat" indication of the health of an application. This heartbeat may be ascertained, for example, by monitoring a total number of database records processed by an application. The heartbeat information may be used, for example, to determine the status, progress, and efficiency of an application being monitored by the monitor module 18. The data may be employed to trigger action by an independent application. For example, where the heartbeat detects that an application has halted processing, the application monitor system 10 may initiate any number of processing including shutting down and restarting the application, including launching recovery applications and systems. The user interface 28 may be used interactively to select for display variable values and to display those selected variable values.

[0054] In some embodiments the client 20 may send the variables 112, 114, including the variable name and variable value, to the user interface 28 in extensible markup language (XML) format. XML supports flexible formatting of the variable names and variable values for transmission to the user interface 28.

[0055] The client and monitor may communicate using a socket connection. When sockets are employed, the monitor module 18 acts as a socket server and the client 20 acts as a socket client. As discussed under the shared memory embodiment of the application monitor system 10, when COBOL is employed as the programming language for encoding the applications 20 and 22 a technical layer may be employed to provide convenient access to operating system socket services.

[0056] The embodiment of the application monitor system 10 where on the monitor module 18 attaches to the memory 112 of the application 12 may be implemented in any

programming language and is not limited to being implemented in COBOL. The applications 12, 22 monitored by the application monitor system 10 may be implemented in any programming language and need not be implemented in COBOL.

General Purpose Computer Systems

[0057] The application monitor system 10 described above may be implemented on any general-purpose computer with sufficient processing power, memory resources, and network throughput capability to handle the necessary workload placed upon it. Figure 4 illustrates a typical, general-purpose computer system suitable for implementing one or more embodiments disclosed herein. The computer system 380 includes a processor 382 (which may be referred to as a central processor unit or CPU) that is in communication with memory devices including secondary storage 384, read only memory (ROM) 386, random access memory (RAM) 388, input/output (I/O) 390 devices, and network connectivity devices 392. The processor may be implemented as one or more CPU chips.

[0058] The secondary storage 384 is typically comprised of one or more disk drives or tape drives and is used for non-volatile storage of data and as an over-flow data storage device if RAM 388 is not large enough to hold all working data. Secondary storage 384 may be used to store programs which are loaded into RAM 388 when such programs are selected for execution. The ROM 386 is used to store instructions and perhaps data which are read during program execution. ROM 386 is a non-volatile memory device which typically has a small memory capacity relative to the larger memory capacity of secondary storage. The RAM 388 is used to store volatile data and perhaps to store instructions. Access to both ROM 386 and RAM 388 is typically faster than to secondary storage 384.

[0059] I/O 390 devices may include printers, video monitors, keyboards, mice, track balls, voice recognizers, card readers, paper tape readers, or other well-known input devices. The network connectivity devices 392 may take the form of modems, modem banks, ethernet cards, token ring cards, fiber distributed data interface (FDDI) cards, and other well-known network devices. These network connectivity 392 devices may enable the processor 382 to communicate with an Internet or one or more intranets. With such a network connection, it is contemplated that the processor 382 might receive information from the network, or might output information to the network in the course of performing the above-described method steps. Such information, which is often represented as a sequence of instructions to be executed using processor 382, may be received from and outputted to the network, for example, in the form of a computer data signal embodied in a carrier wave.

[0060] The processor 382 executes instructions, codes, computer programs, scripts which it accesses from hard disk, floppy disk, optical disk (these various disk based systems may all be considered secondary storage 384), ROM 386, RAM 388, or the network connectivity devices 392.

[0061] While several embodiments have been provided in the present disclosure, it should be understood that the disclosed systems and methods may be embodied in many other specific forms without departing from the spirit or scope of the present disclosure. The present examples are to be considered as illustrative and not restrictive, and the intention is not to be limited to the details given herein, but may be modified within the scope of the appended claims along with their full scope of equivalents. For example, the

various elements or components may be combined or integrated in another system or certain features may be omitted, or not implemented.

[0062] Also, techniques, systems, subsystems and methods described and illustrated in the various embodiments as discreet or separate may be combined or integrated with other systems, modules, techniques, or methods without departing from the scope of the present disclosure. Other items shown as directly coupled or communicating with each other may be coupled through some interface or device, such that the items may no longer be considered directly coupled to each but may still be indirectly coupled and in communication with one another. Other examples of changes, substitutions, and alterations are ascertainable by one skilled in the art and could be made without departing from the spirit and scope disclosed herein.